第６分科会（Ａグループ）

付録 A　リファクタリング時の不具合事例

No.	具体的な不具合の内容を覚えていれば， その内容を記入してください．	不具合の パターン	R-XDDP での 防止	理由
1	改善対象の関数の呼び出し元を洗い出しきれておらず，チームの担当外の機能でデグレードが起こった． レビュー時のメンバーも機能の存在自体を知らなかったため，試験の範囲からも漏れていた． この問題により，修正時には必ず呼び出し元の Grep を行なうことの重要性がチーム内で再認識された．	関数内の機能を不要だと判断し，削除した結果，他プロジェクトでデグレードを起こした	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで防ぐことが可能．ただし，レビューメンバーの選択が重要．
2	改善したが,改善と関わっている気づいてないところでバグが発生してしまった．	関数内の機能を不要だと判断し，削除した結果，他プロジェクトでデグレードを起こした	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで担当者の思い込みを防ぐことが可能．
3	ソフトを 1 本変更したが,変更したクラスが他のソフトでも使用されており他のソフトで不具合が発生した．	関数内の機能を不要だと判断し，削除した結果，他プロジェクトでデグレードを起こした	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで防ぐことが可能． ただし，レビューメンバーの選択が重要．
4	呼び出し元が何十か所もあるような関数の修正をおこなったとき 使用条件が難解（条件がいくつもある）な処理の関数の修正をおこなったとき	関数内の機能を不要だと判断し，削除した結果，自プロジェクトでデグレードを起こした	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで防ぐことが可能． 回答の「難解」「何十か所」という表現から正しくコードを理解できていない可能性が高い．レビューの結果リファクタリングを実施しないと判断するかもしれない．
5	不要となったデバッグ用メッセージを削除したところプログラムの処理タイミングがずれ，デバイス（FPGA）の書き込みに失敗するようになった． しかも,デバイスへの書き込みは一見正しく行われているように見えていた．	関数内の機能を不要だと判断し，削除した結果，自プロジェクトでデグレードを起こした	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで担当者の思い込みを防ぐことが可能．

第６分科会（Ａグループ）

No.	具体的な不具合の内容を覚えていれば，その内容を記入してください.	不具合のパターン	R-XDDP での防止	理由
6	冗長なソースを関数にして簡素化したら,実は冗長と思われていた箇所に意味があってシステムが正常に動作しなくなった，等．まさに元ソースコードの作りの悪さが原因で発生した障害.	関数内の機能を不要だと判断し，削除した結果，自プロジェクトでデグレードを起こした	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで担当者の思い込みを防ぐことが可能.
7	無駄だと判断し，削除したコードが，必要だった．コメントがない為,そのコードが必要な理由がわからなかった.	関数内の機能を不要だと判断し，削除した結果，自プロジェクトでデグレードを起こした	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで担当者の思い込みを防ぐことが可能.
8	フラグの条件が複雑な処理を簡略化した際に,実行毎に条件が変わるのが期待であるのに全く切り替わらなくなってしまった.	機能を変えずに構造を変えるつもりが，機能が変わってしまった	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで，何をしようとしているのか把握することができ，無謀なリファクタリングの場合は止めることが可能
9	複雑な有効無効判断のときに,無効となるはずのパラメータが有効になってしまっていたことがあった	機能を変えずに構造を変えるつもりが，機能が変わってしまった	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで，何をしようとしているのか把握することができ，無謀なリファクタリングの場合は止めることが可能
10	仕様を把握してきれておらずに改善したため,動作が微妙に変わってしまった.	機能を変えずに構造を変えるつもりが，機能が変わってしまった	○	「機能の削除」のプロセス時に変更 3 点セットでレビューすることで，何をしようとしているのか把握することができ，無謀なリファクタリングの場合は止めることが可能
11	別クラスで同じ変数名が多々あり余分に変更してしまい不具合が発生した.	機能を変えずに構造を変えるつもりが，機能が変わってしまった	○	一括で文字列変換していることが原因と思われる． TM で関係ない部分の変更は検出できるため防ぐことが可能
12	必要な判定処理が改善時に漏れてしまう	機能を変えずに構造を変えるつもりが，機能が変わってしまった	○	リファクタリングのパターンにより実施する処理を制限することによって，必要な処理を削ってしまったなどは防ぐことできる

第６分科会（Ａグループ）

No.	具体的な不具合の内容を覚えていれば， その内容を記入してください．	不具合の パターン	R-XDDP での 防止	理由
13	・変数名の間違いにより処理が変わってしまった	機能を変えずに構造を変えるつもりが，機能が変わってしまった	○	変更設計書により何をしようとしているのか確認することで防止できる
14	SQL の SELECT 文でパフォーマンス向上のために処理に不要な項目を削除したが，実際は必要な項目で画面に表示されなくなる不具合が発生した	リファクタリングではない変更によって，機能が変わった	対象外	パフォーマンスの改善なので今回対象としているリファクタリングではない
15	デバイス外部から参照可能にしているデバイス内部データ（プロパティと称す）が複数あり， あるプロパティの参照で発生した不具合を修正した結果，別のプロパティの参照がデグレードした．	リファクタリングではない変更によって，機能が変わった	対象外	不具合の対処方法の問題
16	ある画面の更新処理が間違っていたため修正したが，その画面とほぼ同じ画面がありソースコードが画面ごとに分けられていたため片方だけ更新処理が間違ったままになってしまった．	リファクタリングではない変更によって，機能が変わった	対象外	リファクタリングではなく，機能の改善（機能の変更）の実施時の不具合
17	改造対象と同じような機能の関数が二つあり，それぞれ別に呼び出されている箇所があったが，一方だけを改造し，もう一方を見落としてしまった．	リファクタリングではない変更によって，機能が変わった	対象外	リファクタリングではなく，機能の変更の実施時の不具合
18	一部機能の修正を行ったが，他の箇所でも利用されておりそちらには今回の修正は行われていないかったため問題となった	リファクタリングではない変更によって，機能が変わった	対象外	リファクタリングではなく，機能の改善（機能の変更）の実施時の不具合

付録Ｂ　リファクタリングの実施判断（一部抜粋）

No.	ソースコードの改善の実施可否判断の理由をお答えください
1	機能追加時にソースコード見直しを行ったため，改善の影響範囲を考慮してソースコードの改善を行ったことがある．
2	動いているコードに手を入れることになるので，目の届かない部分で発生するデグレが心配． 目が届いていないため，テストからも確認項目として漏れてしまう．
3	すでにリリースしている機能であれば，修正の工数と影響を考慮し，なるべく修正しないで済むようにする．新規機能で既存機能への影響が少ないものあれば，工数と工程による．
4	改善することで，今後のメンテナンス（バグ発生含む）に効果があると判断したため．
5	その時に余裕があるかどうか（工数・納期等）
6	ソースコードの改善を行うと，その分のコードレビューや評価の工数が発生するため，問題のないソースコード（特にリリース済のコード）であれば改善はしない．
7	改善することで大改造になりかねないため．
8	改善する必要があった時に，今行っているものや，先のスケジュールなどを見て，その改善が行えるか，行えた場合にどのくらいの効果があり，他のスケジュールに影響を与えてまで行うほどの効果があるかによって，判断するかを相談して考えます．
9	改善する意味があるかどうか
10	改善に費やす工数が多いので，次々とアイテムが追加される中で改善することは難しい
11	エンドユーザーに影響が高いものは顧客満足度を向上させることができ，再度依頼を受けられるなどの営業利益に貢献できる．また，コードの改善で可読性向上や修正ステップ数の減少等で業務効率化を図ることができるため．
12	改善するべきコード規模を判断した．
13	（リリース済みの製品のソースの場合）影響範囲が広いとバグを作り込む可能性があるため変更できない
14	・修正もれによる不具合の発生を未然に防げるか？ ・今後の仕様追加があっても，リアルタイム性を担保できるか？
15	改善する事が容易なもの，影響範囲が判断できるものは改善するが， 影響範囲が広がり，テストする工数が確保できないと判断すると改善しない．
16	改造しても，ソフトウェアの振る舞いに影響がない，と判断できた場合のみ行う
17	時間との兼ね合いにもよるが，影響の範囲が特定できない又は影響範囲が膨大すぎる場合は改善しない場合が多い．
18	動作に問題がある場合は改善を行う．すでに動作しており問題がなければ基本的には変更は行わない．
19	費用対効果
20	改善が今回の目的に波及するのか，もしくは，改善しないと現在・将来にバグ発生に繋がるのかを判断したで改善するどうかを考える
21	改善により明らかに可読性の向上，バグの作りこみリスクの低減が見込め，かつ工程に余裕がある
22	改善による評価工数の増大や副作用によるバグの発生．

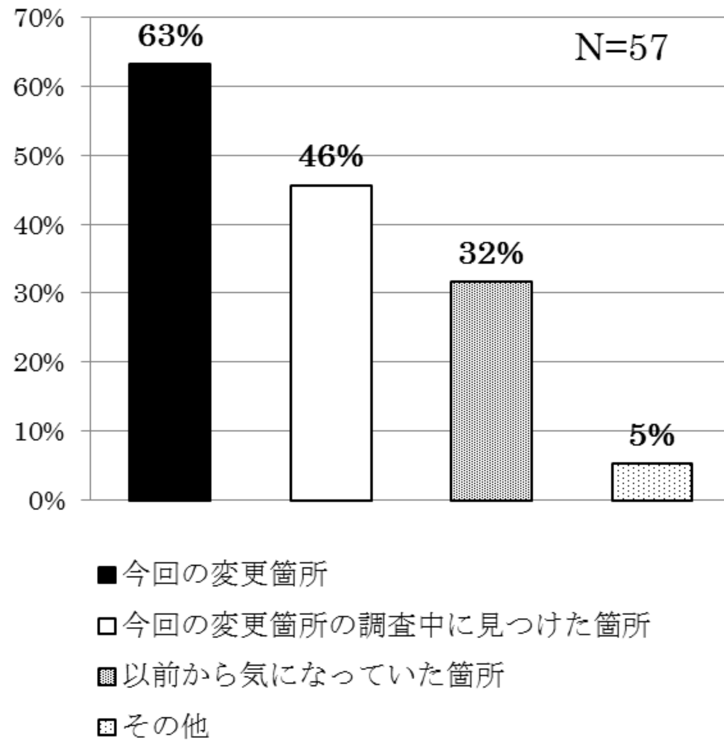
第６分科会（Ａグループ）

No.	ソースコードの改善の実施可否判断の理由をお答えください
23	今後の保守性
24	変更に対してテストの工数などを含め完成までにかかる時間と〆切までの時間から判断します.
25	デグレードの危険性を第一に考え，システムに深刻な問題を起こし得ないような, 影響範囲の浅い・狭いものを優先する．その際，改善する関数の呼び出し元を全てGrep等で洗い出す. 改善の工数・効果も併せて考える.
26	改善した際にデグレードの範囲が大きい場合，試験工数が多くかかりすぎるため.
27	改善を行えば，ミスが少なくなるため.
28	改善の影響範囲が大きければすなわちバグを生み出してしまう可能性も高くなると予想されるため
29	基本的には既存のコードに手を加えるべきではないという判断から，ソースコードの改善は追加機能の修正範囲内で行うこと.（ソースコードの改善だけによる作業工数があることはほとんどない）
30	工数に余裕があればまとまった時間をとってソースコードの改善を行うが，実際には工数の余裕が無い.
31	改善の影響範囲によっては，リリース時に変更点が多くなりテスト工数が大きくなるため影響がでかくなる部分については現状のままにする.
32	既に稼働しているシステムのソースコードに手を入れる事になるので，それを行う事で発生するリスクを検討する.（検討した結果，そのままにしておく事が多い）
33	このまま放置していたら，次回そのソースコードを見る方が，自分と同じような状況になると思ったため. また，しばらくした後に自分が見てもまた同じような状況になってしまうと思ったため. やれるときに改善はするべきだと判断した.
34	当初予定していた作業に追加となるため，改善にかかる調査・開発の時間や既存機能の再確認などが必要になる. また，現状で動作している機能に手を加えることによる不具合の発生する可能性を考えると，十分に時間をかけて対応できるときにしか行えない.
35	無い袖は振れない為.
36	4時間程度の工数で，修正漏れのリスクを軽減できるのであれば，積極的に改善を行うべきと考えた.
37	原則，気が付いたときにやらないとやらないと考えているため. （もちろん規模が大小によっても判断は異なる）
38	改善の目的は保守性の向上による品質向上. 実装工程までに問題を見抜けば随時改善する. 単体試験後は，その改修に限定した影響範囲，品質保障が正しく評価できないため改修しない. 改修する場合においても既存の作業工程には混ぜない．工程完了後の品質を担保した後, 改修作業として別途工程を組み直す.
39	・優先的に考慮するのは費用対効果．工数に見合う効果が見いだせなければ，優先度が低くなる.
40	何年も派生開発してきた案件だったため，使用されていないソースコードが大量にありました. リリース物件のスリム化という意図もありソースコード改善につながりました.
41	改善による影響範囲が大きくなるとテスト工数も大きくなり，期限内に納まらなくなる可能性があるため.

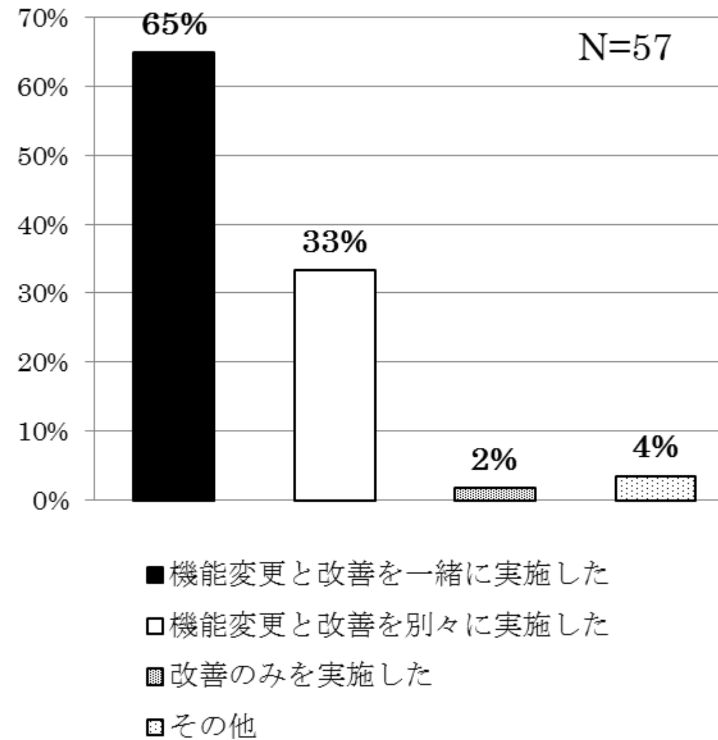
第６分科会（Ａグループ）

No.	ソースコードの改善の実施可否判断の理由をお答えください
42	コーディング規約違反については，修正対象箇所にそれがあれば直す，あるいは，全体見直しにより直す．それ以外(冗長なコードの一元化)などについては，基本的には改修箇所以外は変えない．しかし，影響範囲，改修との兼ね合い（改修規模が大きいので全体的に試験やり直しになる，どうせやるならここも直したほうが逆に効率が良いだろう等）を考慮した上で直すこともある．
43	影響範囲が大きくなる場合，コード修正によりバグが発生する可能性が高い
44	改善を行うことでその後に既読性がよくなり不具合発生時にも解析が容易になるかどうかを判断して決める．
45	簡単に改善できそうか否かで，簡単であればすぐに修正する． いずれにせよ処理の調査をする時に周りのソースコードは眺めるので，そのついでに改善できる所は改善してしまう． また，改善のための工数云々を考えている時間があったら，直してしまったほうがよい．
46	改善策を取ったことによってデグレードが起こされないよう，確実に大丈夫だという確信があれば 改善したほうが良いと判断します．が，作成者がそれなりのキャリアを持つ人間だったりすると， 「こうしているのは，何か別の深い意味があるのかもしれない」といった，熟慮が必要になってしまうので，そういった場合は工数の都合も考えて， 二の足を踏んでしまう状況ですね・・・．コメントで残しておいて貰えれば，理解も早くなるのですが・・・．
47	改善するか否かは改善することによって生じる試験の工数が大きく影響すると思います．改善した場合，元のソースコードが悪ければ悪いほど試験の量も増大します．そのため納期のことを考慮すると，重大なものほど改善しないケースが多いと感じます．
48	改善を行う時間（工数）を取ることができるか．
49	上司・顧客先の判断
50	・このまま作成した時明らかに保守性が下がるとき
51	1．基本的には改善の為の工数は確保されていない為，作業進捗との調整となる． 　ただし，不具合の修正等と合わせて対応可能な場合は，修正後の確認作業が重複しない為， 　影響範囲見合いで修正する場合もある． 3．改善することで処理速度や，拡張性，可読性が上がると判断された場合． 4．工数と同様．
52	現状は問題ないが，関連するソースを修正すると不良になる可能性があったため．
53	・本来の開発工数の誤差程度の範囲で改善が可能かどうか． ・影響範囲は，本来の開発の影響範囲内かどうか． ・稼働実績はどの程度あるか．
54	納期が存在するため，納期に間に合わせることが優先してしまう． 改善しなくても動く結果が出ているソースならばそのままで行く事が多い．まだ工程があまり進んでいなく， 改善している時間があるならば改善を行う．

付録Ｃ　リファクタリングの実態調査アンケートの結果



付図 C-1:ソースコードの改善対象



付図 C-2：ソースコード改善の実施形態

付録Ｄ　「管理関数」と「処理関数」による関数の分割のルール

本資料は，「リアルタイム・システムの構造化分析」での「P-Spec」「C-Spec」の考え方[5]に基づいた関数分割の方法を“「管理関数」と「処理関数」のルール”としてまとめたものである．

【定義】

処理関数：データの処理をおこなう関数（理想は機能凝集度）

管理関数：管理関数と処理関数の管理や制御をおこなう，データの処理はおこなわない．
　　　　　if 文や，switch 文など条件分岐の制御はおこなう．

【ルール】

管理関数は，管理関数と処理関数を呼び出すことができる．

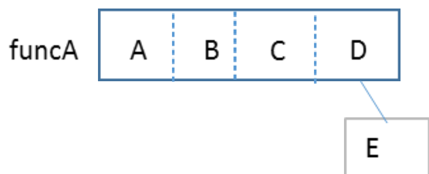
処理関数は，管理関数に呼び出されるのみである．ただし，共通関数のみを呼び出すことができる．

【処理関数と管理関数に分けるメリット】

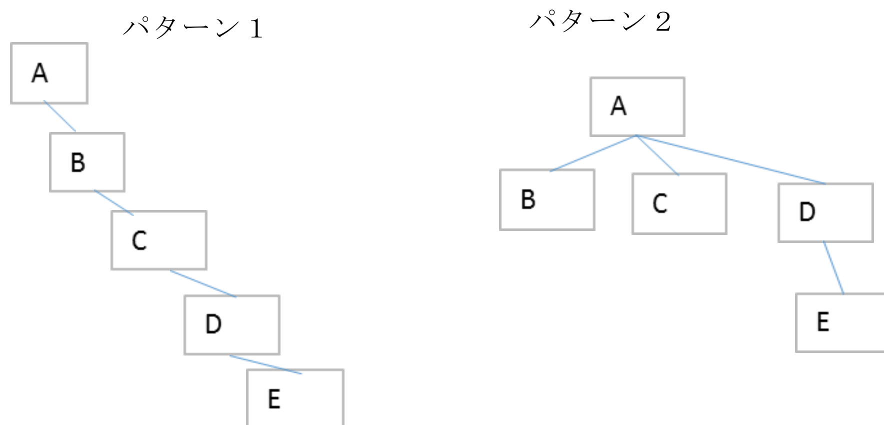
１）処理関数が出てくれば終端であることがわかる．

２）分割結果が一種類になる

大きな関数を，次の A～E に分割する場合：
（E は funcA から呼び出されている関数）



処理関数と管理関数のルールがない場合：分割のパターンがいろいろある



処理関数と管理関数のルールがある場合：分割のパターンが１つ